Preface

The Analysis of Algorithms is a core Computer Science area which provides information on the expected, i.e. the average-case, performance of algorithms. Such information is useful in a variety of applications, including power estimation and resource budgetting in a real-time context. The Analysis of Algorithms also provides fundamental insights in the design of efficient software. Hence, both from an applied and a theoretical perspective, the investigation of improved methods and tools for static average-case analysis is a worthwhile goal.

Average-Case Analysis involves a variety of techniques which, typically, do not allow for automation. Currently algorithms must be analyzed on a case-by-case basis and it is not feasible in general to statically derive the average number of basic steps carried out by an algorithm during its execution. Various bottle-neck problems have been high-lighted in the literature and some well-known algorithms escape analysis.

In view of the status of the field, the ultimate aim to provide a unified foundation for average-case analysis motivated the work of many authors including [Knu73, FS95, Ram96, Vui80]. As pointed out in [Vui80]:

> A progress in our understanding of these questions should drastically affect the way in which we discover and explain the fundamental algorithms, as catalogued by Knuth [Knu73] and Aho et al [AHU87].

The aim of this work is to present a new approach to the Average-Case Analysis of Algorithms, based on the novel notion of random bags and their preservation. The view presented here is that the notion of a random bag may serve as a unifying model for abstract data structures and their data distribution, while random bag preservation enables the constructive tracking of the distribution during computations. The approach inspired novel algorithms and considerably simplified their average-case analysis.

The work presents a modular calculus for static average-case analysis which drastically simplifies the analysis and opens up the way for novel explorations on static timing tools. Random bags also contribute a visual way to represent data and their distributions, which, in addition to facilitating average-case analysis, provides a useful teaching aid.

A parallel between the role of Static Analysis in Software Engineering and the role of Calculus in "real" Engineering may be helpful to illustrate the motivation behind the research. Engineering offers the capacity to analyse the strength of a construction, such as a bridge, by analyzing its blue prints, rather than subjecting it to heavy loads to test its limits. This approach should ideally find a natural parallel in Software Engineering via Static Program Analysis. Rather than executing a program on a large selection of inputs to experimentally derive information on its average-case behaviour, the goal is to derive this information statically via an analysis of the program's source code. Calculus supports the analysis of blue-prints in Engineering. Similarly, the aim of this work is to provide a foundation for a Calculus supporting Static Average-Case Analysis of a program's source code. This is a major challenge and our aim is not to provide an all-encompassing answer. Instead, we focus on the introduction of new advances in this area as a basis for a simplified and unified theory of average-case analysis and as a potential platform on which to build future improved modular static analysis tools.

A central aspect of the novel approach, which distinguishes it from prior approaches to Average-Case Analysis, is the use of randomness preservation to ensure the compositionality property, well-known from the Semantics and the Real-Time Language areas, in the context of the Analysis of Algorithms. Compositionality can rightfully be referred to as the "golden key" to static analysis, witnessed by its central role in static worst-case time analysis. A main theme of this work is that compositionality, combined with the capacity for tracking data distributions, unlocks a novel technique for modular average-case analysis. This approach provides the inspiration for the $\mathcal{MOQA}$[1] "language". The language essentially consists of a suite of random bag preserving data-structuring operations together with conditionals, for-loops and recursion and hence can be incorporated in *any* traditional programming language, importing all of its benefits in a familiar context[2].

A key feature of $\mathcal{MOQA}$ is that its operations have been purpose designed to ensure the capacity for a compositional static average-case analysis of $\mathcal{MOQA}$ code. The guaranteed compositionality property of $\mathcal{MOQA}$ programs brings a strong advantage for the programmer. The capacity to combine parts of code, where the average-time is simply the sum of the times of the parts, is a helpful advantage in static analysis. Moreover, re-use is a key factor in our approach: once the average time is determined for a piece of code, then this time will hold in any context. Hence it can be re-used and the timing impact is always the same. Compositionality also improves precision of static average-case analysis, supporting the determination of accurate estimates on the average number of basic operations of programs.

It is a main theme of the current work to introduce the new foundation for average-case analysis and to illustrate its applicability, as well as to motivate and specify the $\mathcal{MOQA}$ language and discuss its associated static average-case timing tool *Distri-Track*.

[1] MOdular Quantitative Analysis.

[2] $\mathcal{MOQA}$ is implemented at CEOL in Java 5.0 as $\mathcal{MOQA}$-Java.

The work is carried out at the intersection of several areas: Analysis of Algorithms and Random Structures, Semantics, Real-Time Languages, Static Program Analysis, Modular Design and the mathematical theories of Finite Partial Orders, Linear Extensions, Multi-Sets and Probability Theory. Hence the material may be useful for a variety of researchers and students, with interests in Computer Science, Electrical Engineering or Mathematics.

We provide an overview of the chapters in this work.

Chapter 1 provides an introduction to the new techniques for average-case analysis and focuses on a motivation of the central notions involved. This includes a motivation of compositionality as the "golden key" to static timing and the need for novel language design to reach compositionality, including the related concept of an Efficiency-Oriented Language.

The chapter provides a brief introduction to the $\mathcal{MOQA}$ language, for which static average-case timing can be achieved in a modular way through the tracking of distributions. Random bags are introduced as concise ways to capture data and their distribution and distribution tracking is incorporated via the concept of random bag preservation. The split operation, well-known from algorithms such as Quicksort and Quickselect, is provided as an example of a random bag preserving operation. This example also serves to illustrate the tracking of distributions in $\mathcal{MOQA}$ and the use of the notion of a separative function to establish random bag preservation.

The chapter also discusses the central Linear-Compositionality Theorem, which forms the basis for the static derivation of the average-case time of $\mathcal{MOQA}$ programs. Advantages of the $\mathcal{MOQA}$ approach are outlined and the chapter concludes with a discussion of the related area of bridging Semantics and Complexity and the area of Real-Time Languages.

Chapter 2 presents introductory notions, including partial orders, series-parallel orders, trees, heaps and bags. A brief overview of some basic sorting algorithms is provided as well as an introduction to standard timing measures, including exact time, total time, worst-case, best-case and average-case time.

Chapter 3 introduces the central notion of compositionality, including IO-compositionality. Worst-case time is shown to be semi-IO-compositional while average-case time is shown to be IO-compositional. The Average-Case Time Paradox is discussed in this context. This paradox regards the fact that even though average-case time is shown to have better compositionality properties than worst-case time, in practice the derivation of average-case time is known to be much more difficult than worst-case time. The paradox is shown to be linked to the potential lack of randomness preservation of standard algorithms, including well-known examples such as Bubblesort and Heapsort. Moreover, the chapter motivates how IO-compositionality of the average-case time measure can be used, in combination with randomness preservation, to obtain linear-compositionality. This greatly facilitates average-case time analysis and overcomes the Average-Case Time Paradox.

Chapter 4 revisits in a slightly more general context, the fundamental notions of random structures, random bags and their preservation, which have been introduced in Chapter 1. The State Theorem is presented which enables an interpretation of states in random structures as “generalized permutations”. Chapter 4 also introduces the central notion of an isolated subset. An isolated subset forms a subset of a partial order such that the restriction of the random structure over this partial order to the isolated subset is guaranteed to yield a new random structure. A simplified definition of an isolated subset is obtained for the case of series-parallel orders. The chapter concludes with the Extension Theorem, which demonstrates that it is sufficient to define random bag preserving operations locally on an isolated subset, where the extension of the operation to the entire random structure is obtained in a natural way.

Chapter 5 introduces the basic $\mathcal{MOQA}$ operations, including the Random Product, the Random Deletion and Percolation, the Random Projection, the Random Split and the Top and Bot operations. Each of these $\mathcal{MOQA}$ operations is shown to be random bag preserving. Deletion operations typically are not included in the context of automated average-case analysis, since the analysis of deletions with respect to average-case time is well-known to be problematic, even in the context of traditional average-case analysis. Hence the Random Deletion opens up the way for the inclusion of novel algorithms, such as Percolating Heapsort and Treapsort, which are analyzed in Chapter 9. The Extension Theorem of Chapter 4 is applied to extend these operations from local applications on isolated subsets to applications over the entire random structure. Uniformly random bag preserving operations are singled out as of particular interest, since this type of operations enables simplifications of probability computations in later chapters. The $\mathcal{MOQA}$ operations are shown to preserve series-parallel data structures which yields a characterization of the so-called $\mathcal{MOQA}$ atomic-constructible data structures as series-parallel orders. Finally, some simplifications for the series-parallel case are obtained in the context of the computation of cardinalities of random structures. Such simplifications for series-parallel orders will also be useful in the context of Chapter 6, which regards the average-case analysis of the basic $\mathcal{MOQA}$ operations.

Chapter 6, joint with D. Early, presents the detailed average-case analysis of the basic $\mathcal{MOQA}$ operations, resulting in the formulas obtained by D. Early. As shown in Chapter 7, $\mathcal{MOQA}$ programs are Linearly-Compositional with respect to the average-case time, i.e. their average-case time can be expressed as linear combinations of the average-case times of more basic components. Hence, ultimately, a successful average-case time derivation yields the average-case time of $\mathcal{MOQA}$ programs, expressed in terms of the average-case times of the basic $\mathcal{MOQA}$ operations. Formulas for the average-case times of basic $\mathcal{MOQA}$ operations are obtained in Chapter 6 and simplified formulas are derived for the case of series-parallel orders. These formulas are systematically applied in Chapter 9, which presents examples of compositional average-case time derivations of $\mathcal{MOQA}$ programs. Finally, the formulas of this chapter are illustrated via basic applications involving inductively defined data structures, such as linear orders and complete binary trees. Chapter 6 concludes with a demonstration of combinatorial identities used in the derivation of the average-case time formulas.

Chapter 7 provides the specifications for the $\mathcal{MOQA}$ language, with special attention given to conditionals and recursion, which typically form a challenge for static timing analysis. The random bag preservation of $\mathcal{MOQA}$ programs is demonstrated and the method for the linear-compositional derivation of the average-time of $\mathcal{MOQA}$ programs is outlined.

Chapter 8 provides examples of well-known sorting and search algorithms implemented in $\mathcal{MOQA}$. It also includes examples of two novel algorithms, Percolating Heapsort, the first randomness preserving version of the Heapsort algorithm, and Treapsort, a sorting algorithm over treaps; both of which are essentially based on the Random Deletion operation of Chapter 5.

Chapter 9 provides the compositional average-case time derivation of the programs discussed in Chapter 8, with a main focus on illustrating the use of random bags in this context. The chapter in particular presents the first exact average-case time analysis of a heapsort variant via an analysis of Percolating Heapsort. Compositional average-case time derivations, whenever appropriate, rely on the formulas obtained in Chapter 6. The derivations obtained in this chapter illustrate the basic techniques involved in the static timing tool *Distri-Track*.

Chapter 10, joint with D. Hickey and M. Boubekeur, discusses in more detail the static timing tool *Distri-Track*, developed by D. Hickey. *Distri-Track* analyses $\mathcal{MOQA}$ algorithms programmed in Java, using an implementation of $\mathcal{MOQA}$ by J. Townley. *Distri-Track* enables the automated static derivation of average-case time of most of the $\mathcal{MOQA}$ programs presented in Chapter 8. Experiments, including comparisons with time derivations relying on a Java profiler, are discussed, as well as potential implications for Real-Time Languages. Finally, Chapter 11 presents the conclusion and some potential future work.

The book is accompanied by a software tutorial "Static average-case analysis of programs: a beginner's guide to successful tracking". The tutorial requires Adobe Flash Player, which is freely available online at http://www.adobe.com/. The tutorial provides an introduction to the main concepts used in this work as well as videos illustrating the basic $\mathcal{MOQA}$ operations and a selection of $\mathcal{MOQA}$ programs. The reader is advised to read Chapters 1 and 3, followed by a viewing of the tutorial, before proceeding with later chapters in this work.

Cork, Ireland, December 2007 *Michel Schellekens*

Chapter 2
Introductory Notions

We denote the set of real numbers by $\mathcal{R}$ and the natural numbers by $\mathcal{N}$.

We use the following notation for the combinatorial choice of k integers out of l, without repetitions: $\binom{l}{k}$. We denote the number of choices of k_1 integers out of l, followed by k_2 integers out of $l - k_1, \ldots$, followed by k_n integers out of $l - \sum_{i=1}^{n-1} k_i$ by $\binom{l}{k_1,\ldots,k_n}$, where

$$\binom{l}{k_1, \ldots, k_n} = \binom{l}{k_1}\binom{l-k_1}{k_2}\ldots\binom{l-\sum_{i=1}^{n-1} k_i}{k_n}.$$

The cardinality of a set X is denoted by $|X|$. The *range* of a function $f\colon X \to Y$ is the set $\{f(x)|\, x \in X\}$, denoted by $Ra(f)$. The result of restricting a function to a subset A of X is denoted by $f \restriction A$. Similarly, the restriction of a partial order $(X, \sqsubseteq)$ to a subset A of X is denoted by $(A, \sqsubseteq\restriction A)$ or often, when no confusion can arise, by $(A, \sqsubseteq)$.

For any finite collection of sets, these sets are said to be *pairwise disjoint* in case any two distinct sets in the collection are disjoint, i.e. a collection of sets $A_1, \ldots, A_n$ is pairwise disjoint in case $\forall i, j \in \{1, \ldots, n\}.\, i \neq j \Rightarrow A_i \cap A_j = \emptyset$.

The result of concatenating two sequences, say $A = (a_1, \ldots, a_m)$ and $B = (b_1, \ldots,$
$b_n)$ is the sequence $Conc(A, B) = (a_1, \ldots, a_m, b_1, \ldots, b_n)$. Concatenation of more than two sequences, $Conc(A_1, \ldots, A_n)$, is defined in a similar way.

An *affine combination* of a sequence of real-valued functions $f_1, \ldots, f_n$ is an expression of the form $\alpha_1 f_1 + \ldots + \alpha_n f_n$ where $\alpha_1, \ldots, \alpha_n$ are real numbers and $\sum_{i=1}^{n} \alpha_i = 1$.

2.1 Partial Orders & Hasse Diagrams

A *partial order* is a pair $(X, \sqsubseteq)$ consisting of a set X and a binary relation $\sqsubseteq$ between elements of X such that the relation is:

1) Reflexive: $x \sqsubseteq x$
2) Transitive: $x \sqsubseteq y$, $y \sqsubseteq z \Rightarrow x \sqsubseteq z$.
3) Anti-symmetric: $x \sqsubseteq y, y \sqsubseteq x \Rightarrow x = y$.

Unless stated otherwise, we will only consider finite partial orders in the following. For $x \in X$, we let $x{\downarrow} = \{y |\, y \in X$ and $y \sqsubseteq x\}$ and $x{\uparrow} = \{y |\, y \in X$ and $x \sqsubseteq y\}$. If $A \subseteq X$ then $A{\uparrow} = \cup_{x \in A} x{\uparrow}$ and $A{\downarrow} = \cup_{x \in A} x{\downarrow}$.

If $(X_1, \sqsubseteq_1)$ and $(X_2, \sqsubseteq_2)$ are partial orders then a function $f\colon X_1 \to X_2$ is *increasing* iff $\forall x, y \in X_1 . x \sqsubseteq_1 y \Rightarrow f(x) \sqsubseteq_2 f(y)$. In case the function f is an increasing bijection and f^{-1} is increasing, we refer to f as an *order-isomorphism* between the given partial orders.

If $(X, \sqsubseteq)$ is a partial order then $x \sqsubset y \Leftrightarrow (x \sqsubseteq y$ and $x \neq y)$. The binary relation $\sqsubset_1$ is defined to be the set of all pairs (x, y) such that $x \sqsubset y$ and $\nexists z. x \sqsubset z \sqsubset y$.

We assume that the reader is familiar with Hasse diagrams [DP90] which will be used to represent partial orders in the examples. The transitive reduction of a partial order $(X, \sqsubseteq)$ is obtained by omitting from $\sqsubseteq$ all its reflexive pairs and pairs that can be inferred by transitivity. The Hasse diagram of a partial order is a digraph representation of its transitive reduction, where one requires that in the representation, related elements x, y where $x \sqsubseteq y$ are displayed such that x is below y in the Hasse diagram. In other words, Hasse diagrams represent directed acyclic graphs for which the transitive reflexive closure is the given partial order. In Hasse diagrams one only displays the relation $\sqsubset_1$. When specifying a finite partial order we typically list a set of pairs specifying the Hasse diagram for the partial order.

A *linear order* $(X, \sqsubseteq)$ is a partial order such that every pair of elements $x, y \in X$ is related with respect to this order, i.e. $\forall x, y \in X. x \sqsubseteq y$ or $y \sqsubseteq x$. For any set X, the *discrete partial order* on X is the partial order consisting only of the reflexive pairs $\{(x, x) |\, x \in X\}$.

For any subset Y of a partial order $(X, \sqsubseteq)$ we say that Y is a *discrete subset* of the partial order in case the restriction of the order $\sqsubseteq$ to Y is the discrete order.

An element x of a partial order $(X, \sqsubseteq)$ is *maximal (minimal)* iff $\nexists y \in X. x \sqsubset y$ $(y \sqsubset x)$. An *extremal element* of a partial order is an element which is maximal or minimal. A *maximum (minimum)* of a partial order $(X, \sqsubseteq)$ is an element $x \in X$ such that $\forall y \in X. y \sqsubseteq x$ $(x \sqsubseteq y)$.

For any two points $x, y \in X$, a sequence $(x_1, \ldots, x_n)$ is a *path* from x to y when $x_1, \ldots, x_n \in X, x_1 = x, x_n = y$ and $\forall i \in \{1, \ldots n-1\}. x_i \sqsubseteq x_{i+1}$ or $x_i \sqsupseteq x_{i+1}$. A *connected partial order* $(X, \sqsubseteq)$ is a partial order such that for every two points x, y of X there exists a path from x to y. A *component of a partial order* is a non-empty connected subset of maximal size. Any partial order can be partitioned (as a graph) into a set of components. The *length of a path* of a finite partial order is the number of elements on the path. A *chain in a partial order* $(X, \sqsubseteq)$ is a path which forms a linear order under the restriction of $\sqsubseteq$.

2.2 Series-Parallel Orders

We recall some main results regarding series-parallel partial orders [VTL79, Gra81, Gis88, Fin03].

Definition 2.1. Given two disjoint partial orders $(P, \sqsubseteq_1)$ and $(Q, \sqsubseteq_2)$.
The *sequential composition*, denoted via a semi-column ";" is the partial order $P; Q$ on $P \cup Q$ such that $x \sqsubseteq y$ in $P; Q \Leftrightarrow$

$$[x, y \in P \text{ and } x \sqsubseteq_1 y] \text{ or } [x, y \in Q \text{ and } x \sqsubseteq_2 y], \text{ or } [x \in P \text{ and } y \in Q].$$

The *parallel composition* $||$ is the partial order $P \,||\, Q$ on $P \cup Q$ such that $x \sqsubseteq y$ in $P \,||\, Q \Leftrightarrow$

$$[x, y \in P \text{ and } x \sqsubseteq_1 y] \text{ or } [x, y \in Q \text{ and } x \sqsubseteq_2 y].$$

A *series-parallel partial order (SP-order)* is a partial order that can be recursively constructed by applying the operations of sequential and parallel composition starting with a single point [Stan99].

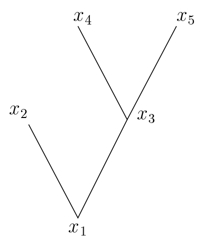
Note that trees form an example of SP-orders as illustrated in Example 2.1 below.

Remark 2.1. For reasons which will become apparent later on, we will refer in the remainder of this work to the sequential composition as the *product operation*. Hence we will use the notation $\bigotimes$ for sequential composition in the following. The parallel composition of two partial orders in our context will amount to a reference to *components* of partial orders.

From the definition of an SP-order it is clear that each SP-order over a finite set, say $\{x_1, \ldots, x_n\}$, can be represented through a formula from the following inductively defined collection $\mathcal{SP}$, where $\mathcal{SP} = \cup_{n \geq 0} \mathcal{SP}_n$ and:

$$\mathcal{SP}_0 = \{x_1, \ldots, x_n\}$$
$$\forall n \geq 1. \mathcal{SP}_n = \{[y \bigotimes z] \,|\, y, z \in \mathcal{SP}_n\} \cup \{[y \,||\, z] \,|\, y, z \in \mathcal{SP}_n\}.$$

Example 2.1. Consider the following SP-order, which corresponds to the formula $[x_1 \bigotimes [x_2 || \, [x_3 \bigotimes [x_4 || x_5]]]]$.



Remark 2.2. Note that in the above approach, there is a unique parsing of each SP formula from $\mathcal{SP}$. However, through associativity, one can drop brackets and focus on the "parallel components" and the "series components" as discussed below.

An SP-order $(P, \sqsubseteq)$, after removing unnecessary brackets, is of one of the following forms:

1) $P = P_1 || \dots || P_n$ *or*

2) $P = P_1 \bigotimes \dots \bigotimes P_n$.

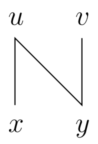
Definition 2.2. We refer to an SP-order in Remark 2.2 of the first kind as a *parallel SP-order* and to an SP-order of the second kind as a *product SP-order*. Moreover, we refer to the suborders $P_1, \dots, P_n$ of the parallel SP-order $P = P_1 || \dots || P_n$ as *parallel components* and to the suborders $P_1, \dots, P_n$ of $P = P_1 \bigotimes \dots \bigotimes P_n$ as *product components*.

Remark 2.3. It is clear that, by removing unnecessary brackets, we can express a parallel SP-order as an SP-order for which all parallel components are product SP-orders. Similarly we can express a product SP-order as an SP-order for which all product components are parallel SP-orders. We refer to such expressions as *canonical representations* of the SP-order in question.

Example 2.2. Consider the product SP-order $[[x \bigotimes [y \,||\, z]] \bigotimes [z \,||\, [u \,||v]]$, which can be expressed as the product of parallel components $x \bigotimes [y \,||\, z] \bigotimes [z \,||u \,||v]$.

We discuss the following interesting characterization of SP orders as so-called N-free orders.

Definition 2.3. A partial order is *N-free* if there is no quadruple of elements $\{x, y, u, v\}$ whose non-trivial relations are given by $x \sqsubset u, y \sqsubset u, y \sqsubset v$, i.e. there is no suborder determined by a Hasse diagram corresponding to an "N-shape":



The following proposition provides a characterization of SP-orders as N-free partial orders.

Proposition 2.1. [Fin03, Gis88, Gra81, VTL79] *For finite partial orders, the notions of SP and N-free are equivalent.*

2.3 Trees & Heaps

A tree is a partial order with a maximum element, referred to as the *root* of a tree, for which the Hasse diagram representation has no cycles [HJ99, AHU87]. The elements of this order are referred to as the *nodes* of the tree. We follow here the Computer Science convention in considering the root as the largest element in the tree, rather than the standard mathematical approach of defining the root as the least element. The *leaves* are the minimal nodes, *internal nodes* are nodes that are not leaves, a *child of a node*, provided the given node has elements strictly below it, is a node immediately below the given node in the ordering. Similarly, a *parent of a node*, provided the given node has a node strictly above it, is a node immediately above the given node in the ordering [AHU87]. The *depth of a node* in a tree is the number of nodes (not including the given node) on the unique path from the node to the root. The *size of a finite tree* is the number of nodes in the tree.

A *binary tree* is a tree in which every node has at most two children. A *full binary tree* is a binary tree in which every internal node has two children. A *complete binary tree* is a full binary tree in which all leaves occur at the same depth.

In the following we will only consider *finite binary trees* and will simply refer to these as *tree*.

A *near-heap* is obtained from a complete binary tree as follows:

1) Some leaves are allowed to be omitted in right to left order.
2) All nodes of the tree obtained via 1) are labeled with a natural number such that each parent which is not the root has a label which is greater than those of its children.

A *heap* is a near-heap for which the root has a label which is larger than the labels of its children. The set of all heaps of size n, identified up to labeling-isomorphism, is denoted by $\mathcal{H}_n$ and the cardinality of this set is denoted by $h(n)$.

We display the recurrence for the number $h(n)$ of heaps of size n with labelings from a fixed set of n distinct labels [Ede96], where H_1 and H_2 are the two heaps obtained by removing the root of the original heap of size n:

$$h(n) = \binom{n-1}{|H_1|} \times h(|H_1|) \times h(|H_2|).$$

Remark 2.4. For future reference we remark that: $h(1) = h(2) = 1, h(3) = 2, h(4) = 3, h(5) = 8$ and $h(6) = 20$ and thus $h(3)$ does not divide $h(4)$, $h(4)$ does not divide $h(5)$ and $h(5)$ does not divide $h(6)$.

Treaps were introduced independently by several authors [Vui80, SA96], where the first introduction of these structures is due to [Vui80], which used the terminology of *Cartesian Trees* .

A *key* is a value stored at a node x of a tree, denoted by $x.key$. Keys stem from a totally ordered universe and are pairwise distinct. A *Binary Search Tree* consists of a binary tree with a set X of n items stored at the nodes: some item $y \in X$ is chosen to be stored at the root of the tree, and the left and the right children of the root are binary search trees for the sets $X_< = \{x \in X|\, x.key < y.key\}$ and $X_> = \{x \in X|\, y.key > x.key\}$. Binary Search trees satisfy the *in-order* property. This means that for any node x in the tree $y.key < x.key$ for all y in the left subtree of x and $x.keq < y.key$ for all y in the right subtree of x. Let X be a set of items each of which has an associated key and a *priority*. The priorities, as are the keys, are drawn from a totally ordered universe and are assumed to be pairwise distinct. The two ordered universes need not be the same. A *treap* for X is a binary tree with node set X that is arranged in in-order with respect to the keys and in heap-order with respect to the priorities. Pairwise distinctness of keys and of priorities guarantees that there is a unique treap for X [SA96].

2.4 Basic Sorting Algorithms

We give details of the pseudo-code for several algorithms of which the average-case time will be discussed. For further information we refer the reader to [AHU87] and [Knu73].

We will discuss two standard variants of Heapsort in the work, William's version and Floyd's version. We recall some basic background material, to formulate the Heapsort variants, and for simplicity consider heaps labeled with natural numbers.

The traditional Heapsort algorithm relies on a procedure "Push-Down", which transforms a near-heap to a heap. Two versions of Push-Down are available: Williams' original version [Wil64] which we refer to as "*W-Push-Down*" and the more economic version used by Floyd [Flo64], which we refer to as "*F-Push-Down*". We sketch these alternative versions below. For a precise formulation we refer the reader to [AHU87], [Wil64],[Flo64] and [LV93]. In the description given below, "larger" ("smaller") refers to the order $\geq$ ($\leq$), while "strictly larger" ("strictly smaller") refers to the order $>$ ($<$).

W-Push-Down
Given a near-heap of size n, say with root label l, then *W-Push-Down* proceeds as follows:

Start at the root of the near-heap and compare the labels of the two children. If m is the larger label of these two, then compare the root label l with m. If $m < l$, then the algorithm halts since the near-heap is a heap. Otherwise these labels are swapped. Consider the sub near-heap which has as root the child which was originally labeled with m. Recursively repeat this procedure on this sub near-heap until the children of the root under consideration have labels which are both less than the root label or a leaf is reached.

F-Push-Down
Given a near-heap of size n, say with root label l, then *F-Push-Down* proceeds as follows:

Keep track separately of the root label l. Start at the root of the near-heap and compare the labels of the two children. Select the child which has the larger of the two labels. Repeat the procedure on the sub near-heap which has this child as root until a leaf is reached. This determines a chain from the root of the near-heap to the leaf. Move systematically up the chain (if necessary) until a particular node N is found which has a label greater than the root label l.

At that point, assign the label of this particular node N and of each of its ancestor nodes on the chain, to each of their respective parent nodes. That is, move the labels of each node one node upwards along this chain. Finally, change the label of N to the original root label l.

The main difference, regarding comparisons, is the following: *W-Push-Down* proceeds from the root down the near-heap, making two comparisons per step. *F-Push-Down* proceeds from the root down the near-heap to a leaf, making one comparison per step. Then it backtracks upwards along the chain to the root, making one extra comparison per step until a particular node is reached for which the label is larger than the label of the root.

We recall the Heapify procedure which uses Push-Down (where either of the above versions of Push-Down can be selected) to create a heap from a given list. Push-Down(i, j), where $i \leq j$, comes equipped with two parameters which indicate the boundaries i and j of the sub-list being operated on. I.e. i indicates the index of the label to be "pushed down" and j indicates that we only operate on labels of elements with indices ranging from i up to and including j.

A list $(L[1], \ldots, L[n])$ is interpreted by Heapify as a binary tree, with root node labeled with $L[1]$ and such that each node labeled with $L[j]$, with $j \leq \lceil \frac{n}{2} \rceil$ has either two children labeled with $L[2j], L[2j+1]$, when $2j + 1 \leq n$, or one child labeled with $L[2j]$ when $2j = n$ [AHU87]. Push-Down is called recursively in the Heapify procedure defined below. Since Push-Down will initially be called on a binary tree with at most 3 elements, this binary tree is automatically a near-heap and hence the recursive calls to Push-Down are well defined.

Heapify
```
For i = $\lfloor \frac{n}{2} \rfloor$ downto 1 do Push-Down(i,n);
```

The Heapsort algorithm relies on a Selection process, in which the largest label of the heap, i.e. the label at the root, is swapped with the label of the rightmost leaf, after which Push-Down is called once more on the newly created near-heap and the process is repeated.

The pseudo-code for Selection is given by:

Selection
```
For i = n downto 2 do
  swap(L[1],L[i]);
  Push-Down(1,i-1)
```

Finally, traditional Heapsort can be formulated as follows, again using either version of the procedure Push-Down:

Heapsort
```
Heapify; Selection
```

We recall the pseudo-code of the version of Bubblesort discussed in [AHU87], which we refer to as “Bubblesort-I”.

Bubblesort-I
```
For i = n-1 downto 1 do
  For j = 1 to i do
       if L[j] > L[j+1] then swap(L[j],L[j+1])
```

The inner for-loop of Bubblesort-I is denoted by J_i^I where $i \in \{1, \ldots, n-1\}$.

We recall the pseudo-code of the Bubblesort version from [Knu73], which we refer to as Bubblesort-II“”. This version keeps track of the number of swaps performed during a run of the inner for-loop; i.e. of the number of comparisons for which $L[j] > L[j+1]$ is true. In case no swaps occur during this run, the sublist under consideration is sorted and the algorithm terminates.

Bubblesort-II
```
For i = n-1 downto 1 do
  k := 0;
  For j = 1 to i do
       if L[j] > L[j+1] then k := k+1; swap(L[j],L[j+1]);
```

```
If  k=0  then return  L
```

The inner for-loop of Bubblesort-II is denoted by J_i^{II} where $i \in \{1, \ldots, n-1\}$ and $n = |L|$.

2.5 Uniform Distribution and Bags

A *bag*[1] is a finite set-like object in which order is ignored but multiplicity is explicitly significant. Thus, contrary to sets, bags allow for the repetition of elements. Therefore, bags $\{1, 2, 3\}$ and $\{3, 1, 2\}$ are considered to be equivalent, but $\{1, 2, 2, 3\}$ and $\{1, 2, 3\}$ differ. We refer to the number of times an element occurs in a bag as the *multiplicity* of the element. The *cardinality of a bag* is the sum of the multiplicities of its elements. Each bag A of n elements has an *associated set* $B = \{b_1, \ldots, b_k\}$ such that $\cup A = \cup B$ and where each element b_i of B is repeated K_i times where $1 \leq K_i \leq n$ and $\sum_{i=1}^{k} K_i = n$. It is clear that a bag A can be represented in this way as a - set of pairs $\{(b_1, K_1), \ldots, (b_k, K_k)\}$.

In fact it will be convenient to adopt a slight generalization of this type of representation as our formal definition of a bag in the following. I.e. a bag is formally defined in this context as a finite set of pairs $\{(b_1, K_1), \ldots, (b_k, K_k)\}$, where each K_i is a natural number, referred to as the multiplicity of the element b_i and where we do *not* require that the elements b_i are pairwise disjoint. In case $(*)\ \forall i, j.\, b_i \neq b_j$, we refer to the finite set of pairs $\{(b_1, K_1), \ldots, (b_k, K_k)\}$ as a *strict* bag. We allow a more flexible approach in which we do not require $(*)$ to hold since in practice some repetitions of an element b may occur in different contexts, e.g. as K repetitions in one context and L in another, in which case we chose to keep track of these repetitions separately as pairs (b, K) and (b, L) in the same bag rather than as a single pair $(b, K + L)$ in the bag.

To keep track of the number of times that a particular output is produced, we will represent the range of the input-output function of a program as a bag. $\mathcal{MOQA}$-programs are guaranteed to terminate, so there are no undefined outputs to be taken into account.

Notation 2.1 (Input and Output Bag)
For any program P we indicate the bag of its inputs by $\mathcal{I}_P$, referred to as the "input bag". The bag of inputs of size n is denoted by $\mathcal{I}_P(n)$. A *bag of inputs* $\mathcal{I}$ for a program P is a sub bag of the input bag $\mathcal{I}_P$. Typically we will require that $\mathcal{I} \subseteq \mathcal{I}_P(n)$ for some n.

$\mathcal{O}_P(\mathcal{I})$ denotes the bag of outputs, referred to as "the output bag", of the computations of a program P on a bag of inputs $\mathcal{I}$.

[1] Also referred to as multi-set in the literature.

If $\mathcal{I}_P(n) = \mathcal{I}_n$ for a particular data structure under consideration then we denote the output bag $\mathcal{O}_P(\mathcal{I}_n)$ by $\mathcal{O}_P(n)$.

We recall (cf. Remark 1.6) that all programs under consideration are assumed to terminate. Hence it is clear that in case $\mathcal{I}$ is an input bag for a program P, the bags $\mathcal{I}$ and $\mathcal{O}_P(\mathcal{I})$ have the same cardinality where the input-output relation forms the corresponding bijection.

Example 2.3. 1) Consider a sorting algorithm P. The bag of outputs $\mathcal{O}_P(\mathcal{A}_n)$ is $\{(S_n, n!)\}$, consisting of $n!$ copies of the sorted list $\mathcal{S}_n$.

2) Consider Bubblesort-I of Section 2.2 and its inner for-loop J^I_{n-1} for $n = 3$, i.e. J^I_2.
Let $\mathcal{A}_3 = \{(1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1)\}$. The bag of outputs is $\mathcal{O}_{J^I_2}(\mathcal{A}_3) = \{((1,2,3),4),((2,1,3),2)\}$.

Bags are useful to represent sets of data that are uniformly distributed. A bag $A = \{(b_1, K_1), \ldots, (b_k, K_k)\}$ is called *uniformly distributed* iff $\forall i, j \in \{1, \ldots, n\}. K_i = K_j$. It is clear that if $A = \{(b_1, K_1), \ldots, (b_k, K_k)\}$ is a uniform bag then we can simply use the simplified notation $A = \{(B, K)\}$, which indicates that the bag A consists of K copies of the set B. In particular: $|A| = K|B|$. Each element of a uniform bag $\mathcal{A}$ with associated set B arises with equal probability of $\frac{K}{|A|} = \frac{1}{|B|}$.

Example 2.4. Note that Example 2.3, 1) yields a uniform bag, while this is not the case for Example 2.3, 2).

We recall the main rules for computing probabilities for statements involving $\wedge, \vee$ and $\neg$. These will be applied to determine the probabilities of boolean expressions occuring in $\mathcal{MOQA}$ programs.

$$Prob[\neg A] = 1 - Prob[A].$$

For pairwise disjoint event sets $A_1, \ldots, A_n$:

$$Prob[\cup_{i=1}^n A_i] = \sum_{i=1}^n Prob[A_i].$$

The Modularity Law for probabilities two event sets A_1 and A_2:

$$Prob[A_1 \cup A_2] = Prob[A_1] + Prob[A_2] - Prob[A_1 \cap A_2],$$

The Modularity Law for three event sets A_1, A_2 and A_3 :

$$\begin{aligned} Prob[A_1 \cup A_2 \cup A_3] = {} & Prob[A_1] + Prob[A_2] + Prob[A_3] - Prob[A_1 \cap A_2] - \\ & Prob[A_1 \cap A_3] - Prob[A_2 \cap A_3] + Prob[A_1 \cap A_2 \cap A_3]. \end{aligned}$$

The General Modularity Law for event sets $A_1, \ldots, A_n$ $(n \geq 2)$:

$$Prob[\cup_{i=1}^{n} A_i] = \sum_{i=1}^{n} Prob[A_i] - \sum_{i<j} Prob[A_i \cap A_j] + \ldots + (-1)^{n-1} Prob[\cap_{i=1}^{n} A_i].$$

2.6 Timing Measures

We recall the standard definitions of comparison-based algorithms and of worst-case time and average-case time for comparison-based algorithms. We recall from Chapter 1 that a comparison-based algorithm is an algorithm for which every action during the code execution is determined by a comparison between two elements of the input data structure (e.g. [AHU87]). In particular, every assignment and every swap during the execution of the code is a direct consequence of a comparison between two elements. Most sorting and search algorithms fall into this class and traditional lower bound estimates apply in this context.

As indicated in Chapter 1, static timing in our context regards the counting of comparisons during the execution of comparison-based algorithms.

For a comparison-based algorithm P we define the *exact time* $T_P(I)$ on an input I to be the number of comparisons made by the program P during the computation of the output $P(I)$. The notation $T_P(n)$ indicates the restriction of the function T_P to the set $\mathcal{I}_n$. We will consider subsets $\mathcal{I}$ of $\mathcal{I}_n$ and consider the following time measures with respect to $\mathcal{I}$:

The *total time* of P for inputs from $\mathcal{I}$, denoted by $T_P^t(\mathcal{I})$ is defined by:

$$T_P^t(\mathcal{I}) = \sum_{I \in \mathcal{I}} T_P(I).$$

The *worst-case time* of P for inputs from $\mathcal{I}$, denoted by $T_P^W(\mathcal{I})$ is defined by:

$$T_P^W(\mathcal{I}) = max\{T_P(I) | \, I \in \mathcal{I}\}.$$

The *best-case time* of P for inputs from $\mathcal{I}$, denoted by $T_P^B(\mathcal{I})$ is defined by:

$$T_P^B(\mathcal{I}) = min\{T_P(I) | \, I \in \mathcal{I}\}.$$

The *average-case time* of P for inputs from $\mathcal{I}$, denoted by $\overline{T}_P(\mathcal{I})$ is defined by:

$$\overline{T}_P(\mathcal{I}) = \frac{T^t(\mathcal{I})}{|\mathcal{I}|} = \frac{\sum_{I \in \mathcal{I}} T_P(I)}{|\mathcal{I}|}.$$

In order to denote an arbitrary measure, which can include any of the above, we use the notation $\mathcal{T}_P$ and the usual corresponding notations $\mathcal{T}_P(\mathcal{I})$ and $\mathcal{T}_P(n)$.

We observe that:

$$\forall \mathcal{I}. \, T_P^B(\mathcal{I}) \leq \overline{T}_P(\mathcal{I}) \leq T_P^W(\mathcal{I}).$$

If the exact time of P is a constant C on the inputs from $\mathcal{I}$ then:

$$\overline{T}_P(\mathcal{I}) = T_P^B(\mathcal{I}) = T_P^W(\mathcal{I}) = C.$$

For a given data structure, we let the finite set $\mathcal{I}_n$ denote the set of input states of size n for this particular data structure.

Of course, in case $\mathcal{I} = \mathcal{I}_n$, we will for the total, worst-case, best-case and average-case time respectively use the following standard notation based on size indication only: $T_P^t(n), T_P^W(n), T_P^B(n)$ and $\overline{T}_P(n)$.

We assume familiarity with the asymptotic classification of running times and the notion of a decision tree (e.g. [AHU87]). Given two functions $f, g \colon \mathcal{N} \rightarrow \mathcal{R}^+$. Then

$$f \in O(g) \Longleftrightarrow \exists c > 0\, \exists n_0\, \forall n \geq n_0.\, f(n) \leq cg(n).$$

$$f \in \Omega(g) \Longleftrightarrow \exists c > 0\, \exists n_0\, \forall n \geq n_0.\, f(n) \geq cg(n).$$

For comparison-based algorithms one can show that in the asymptotic hierarchy (e.g. [CLR96]) the worst-case time and the average-case time satisfy the following lower bound: $T_P^W(n) \in \Omega(log(N_n))$ and $\overline{T}_P(n) \in \Omega(log(N_n))$ where N_n is the number of leaves in the decision tree of the algorithm P for inputs of size n.